Bioinformatics Crash Course

Release 1.0

Mar 06, 2021

Lessons


UBIC

CHAPTER 1

Lesson 1: Hello Bioinformatics

1.1 Welcome to the Crash Course!

This course is designed for individuals with zero to some programming experience and zero to some bioinformatics experience. It is offered as an in-person course for students at UC San Diego, and some exercises require you to be a student to access our cloud workspace.

Regardless, we hope that the material here is useful even if you are not able to complete all the exercises (and, we hope to make this fully available to everyone soon).

- Sabeel Mansuri and Mark Chernyshev, Founders of the Bioinformatics Crash Course

1.2 Classic Bioinformatics Problems

As you progress through this course (and bioinformatics in general), you might find yourself developing technical skills without knowing what problems they can be used to tackle. In fact, there are several common (and difficult) problems in bioinformatics worth being familiar with. Therefore, this lesson will introduce you to of some of the things we, as bioinformaticians, do.

Note: In this course, we try to avoid lecture-based or text-heavy material. However, in this first lesson, we want to set a foundation of major problems bioinformaticians work on. We'll keep it short, and move on to the interactive part: working in a Linux environment!

1.2.1 I. The Alignment Problem (full lesson)

The first problem we'll look at is alignment. **Alignment is the process of arranging sequences in a way to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.** In order to understand why alignment is hard and practical problem, we will start by learning a bit about sequencing DNA.

Illumina Sequencing:

This is the most widely used "golden standard" method of sequencing DNA. The main idea is that DNA is fragmented into pieces, attached to a flow cell, copied many times over by PCR, and the complement strand is determined one nucleotide at a time. The modified nucleic acids Illumina uses during the generation of complementary strands emit light when they are bound. Illumina sequencing adds one base, measures the color output of the flow cell, adds a different base, measures the color output and repeats over and over. Here is a nice graphic depicting the process:

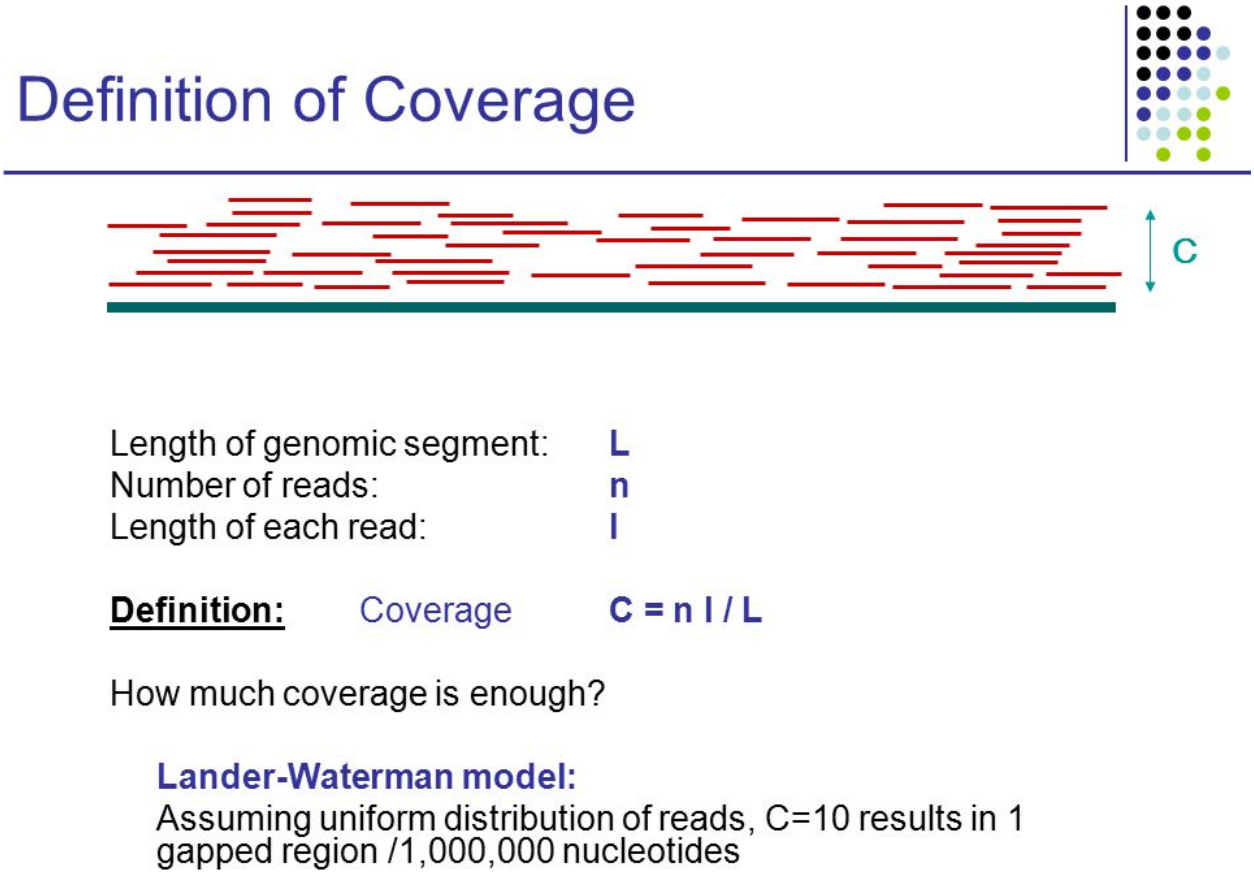
There is also a video explaining Illumina sequencing which you'll watch in *every single Bioinformatics class* you take in the future, found here.

Want to learn more about different sequencing methods? We have a document describing non-Illumina methods in more detail over here

Comparing Sequencing Methods and Why they Matter

What do all sequencing method have in common (yes, even PacBio)? All produce ridiculous quantities of small DNA sequences. Illumina produces 300 million to 4 billion reads per run, with a selection of read lengths ranging from 50 base pairs to 300 base pairs. Meanwhile, Sanger produces 50000 sequences at lengths varying from 800 to 1000 base pairs. To give some perspective, the typical animal of interest is a human and those have 3.0×10^9 base pairs. Individual human genes range from 1148 to 37.7 kb (average length = 8446 bp,s.d. = 7124).

These tiny reads overlap all over the place. If you imagine the true sequence these reads came from and place the reads where they came from, you will get many reads piled up over every base pair in the true sequence. The more reads pile up, the more accurately you can predict the actual sequence. A common measure that rates the robustness of an alignment is coverage:



coverage

Besides helping us reconstruct the DNA put into a sequencer, alignment can also

- Determine which organism an unkown sequence comes from
- Pinpoint locations where mutations have occured relative to a reference sequence
- Help determine the evolutionary distance between sequences

TLDR: There are numerous complex applications of bioinformatics algorithms, from functional structure predictions to ancestral reconstructions. Alignment serves as the foundation for many of these algorithms, making basic sense of the incomprehensible mass of DNA that sequencing gives us.

1.2.2 II. The Clustering Problem (full lesson)

Clustering is the the process of assigning data points to groups in such a way that the elements in a group/cluster are more similar to each other than they are to those in other groups. The definition of what it means to be similar can vary and is determined by the function we use to measure distance between two points.

Some types of clustering:

- Hierarchical Clustering: Repeatedly combines the closest points into a cluster that is the hybrid location of both points. The reason this is interesting to us is that it forms a tree of clusters, which can represent a tree of related genes which can be used to infer homology.

hierarchical cluster

- K-means Clustering: Separates a dataset into k groups of points in such a way that the members of a cluster are as close as possible to the center of the cluster they belong to. This type of clustering can check that the datapoints we are observing cluster together by tissue type, experimental conditions, time points, etc.
- Fuzzy Clustering: Datapoints are not definitively assigned to a specific cluster, rather they are given a likelihood of belonging to a cluster. This can be used to ascertain levels of co expression between genes, revealing genes which may be under common regulatory control.

1.3 Aliview

Alright, enough of the conceptual stuff. Let's get our hands dirty.

Note: The goal of this exercise is not really to master alignment. Instead, it's more to get familiar with using bioinformatics tools and working with bioinformatics data.

1.3.1 Prerequisites

You will need Java to proceed. If you do not have it, go here and press the big red install button in the middle of the page.

Aliview is a sequence viewer with a bunch of built-in tools, including alignment tools. We will use Aliview to see what a typical dataset looks like coming out of the illumina sequencing machine and what it means, visually, to align the sequences. Click here and go to download the stable version for your OS.

1.3.2 Analysis

Next, download this neat dataset. Launch aliview, click file->open file->PC64_V48_small_reconstructed_seqs.fasta. Scroll to the right and notice the mess that begins to form as you scroll.

These sequences are sourced from the same gene and have gone through many cleaning steps, so the differences between them are quite likely to be real (rather than artifacts from sequencing). Click align in the upper left corner and click realign everything. Now scroll forward and observe the gaps that have been inserted by the aligner.

These sequences are actually from HIV-1 glycoprotein envelopes from a person with broadly neutralizing antibodies against HIV-1(sequenced with PacBio). An effective HIV-1 vaccine should evoke the production of broadly neutralizing antibodies, so it is important to study the structure of the envelope that caused these antibodies to develop in individuals. The steps leading up to the data you downloaded allowed us to reveal a few specific strains. Now that we have them aligned, we can start asking questions about the differences between them and their evolution(which is important to figuring out how to counteract them). If you want to go into the nitty-gritty biology behind these sequences, go here. I will make a reconstruction of the evolutionary relationship for you to look at.

The SNP differences are obvious, but you will notice that there are weirder differences - like the >100 bp gaps formed in the middle. The truth is that I do not know why those are there! Those can be

1. Real differences in hypervariable loops
2. non-functional virions
3. RT/PCR artifacts

Here's another cool tool: Blast will quickly look up a sequence in the NCBI database and spit out similar sequences it finds. Now, go to aliview and click edit->delete all gaps in all sequences. Copy the first sequence into the clipboard. Google NCBI Blast and open up the first result. Paste the sequence into the big box in the top of the page and click **BLAST** at the bottom. After a few seconds, Blast should link you to a whole lot of glycoprotein that are... from the article these were published in! This is one of the basic uses of Blast - to figure out where a sequence comes from.

CHAPTER 2

Lesson 2: The Linux Terminal

Bioinformatics is often memory and computation intensive, so we'll be outsourcing our computational work to a bigger computer called a server. This server will run on the Linux operating system. (This is no different than your laptop running on a Windows or Mac operating system.)

Why Linux? The majority of servers run on Linux, a free operating system which inherited its predecessor's (GNU's) mission to give users freedom. Linux is completely open source, allowing users to see and modify any part of its inner workings. Linux is also extremely stable, allowing servers to be up for years at a time without restarting.

2.1 Your First Commands

1. Discover your identity. Type `whoami` into the window that just opened up and hit `enter`. And just like that you're talking with your computer, you bioinformatician, you.

2.2 So... Commands?

Commands are things you can type into the terminal to perform different actions. There are an endless number of commands, each with a ridiculous amount of options, so **do NOT attempt to memorize them on the first try**.

How do you know what command to use to do sometihng you want? Simple: for now, we'll explain commands as you need them. Actually learning (or memorizing) the commands will come naturally from repeated use of the terminal.

Note: Anytime you need a refresher on what a command does, type the command line with the –help option like so: `ls --help`*. If that does not work, try* `man ls`*. One (or both) of these will pull up information on how to use the command. Can you figure out what the ls command does?*

2.3 What Am I Looking At?

When you open up your laptop, you are presented with your "Desktop". When you open up a terminal (or connect to a server via ssh!), you are presented with your "Home".

When you want to open up the "Pictures" folder on your laptop, you find the folder labeled "Pictures", and then open it. Uh-oh... we don't know how to (1) find something or (2) open something!

Let's take a step back and talk about all the files in your computer are organized. As you know, you can have different files stored in different places on your computer. You do this by creating folders (inside of other folders), and creating files inside of them. This is **exactly how our server works**, except we call folders "directories".

Similar to how "Desktop" is a folder on your computer, "Home" is a directory on your account on the server. And similar to how your "Desktop" can have folders created in it, so can "Home". You can look at what's inside with the `ls` command, which is short for "list". Type `ls` to see what's in your "Home" directory. (We now know how to find something!)

2.4 What Can I Do?

There's nothing in your home directory! Let's change that by creating a directory. You can do this with the `mkdir` command. Let's create a directory called "software" by doing `mkdir software`. Confirm that this worked by looking at your home directory again.

Now that we can find the "software" directory, let's move into it (the equivalent of opening a folder on your laptop). Type `cd software` to **c**hange **d**irectory to software. Next, type `ls` and confirm that nothing is in this freshly created directory. Create another directory here called "docs" (yes, directories can contain other directories, much like folders can contain folders!).

Great! You can now make a directory and enter it. The last step is to exit the directory. You can do this by just telling the terminal to change directory to whatever contains the current directory (i.e. the parent directory). The alias for the parent directory is `..` Confirm you've moved back to the home directory using `ls`.

On your laptop, if you want to get to a deeply nested folder, you have to keep unfolding the layers by opening multiple folders. Wouldn't it be nice to be able to just get directly to a folder? We can do this using `cd` by specifying multiple layers we want to change into. For example, we just created a "docs" inside of "software". We can get into this directory by typing `cd software/docs`, where the terminal will recognize "/" as a sign that what follows will be inside of "software".

Finally, let's go back home. You might think that `cd ../..` will take you there, and you would be right! You'd look at the parent's parent, which is the home directory. However, there's an easier way. Much like `..` is an alias to the parent directory, `~` is an alias to your home directory. Simply do `cd ~` from anywhere and you'll end up home!

`.` is a special alias too! Can you figure out what it refers to? Try `cd .` and see where you go.

2.5 [Your Turn!] Investigate Genetic Data

Let's get to work with some real genetic data!

Start in your home directory. Create and enter a directory called "week1". Then run the following command:

```
wget https://raw.githubusercontent.com/biopython/biopython/master/Doc/examples/ls_
↪orchid.fasta
```

You've just downloaded a file full of a bunch of genetic data! You can take a peek at the first few lines by doing `head ls_orchid.fasta`. Your job is to analyze this file.

You have 3 tasks:

1. Print all the contents of the downloaded dataset to terminal window ("standard output")
2. Print how many lines there are in the file

3. Print how many lines there are in the file **THAT CONTAIN GENETIC DATA** (no headers)

The commands cheat sheet below and the hint above about deciphering commands you're not familiar with are your friends. Good luck!

2.6 Commands Cheat Sheet

`ls`(list files) Print out the contents of a directory.

`mkdir`(make directory) Create a directory with the same name as the argument you give it.

`cd`(change directory) Change directory to whatever is specified.

`head` Print the beginning of the specified file to the terminal.

`cat` Print whatever follows to the terminal. If a file name is specified, print the contents to the terminal.

`wc` (word count) Print the number of words in the file name specified after the command.

`grep` Print out lines matching the specified conditions.

CHAPTER 3

Lesson 3: Advanced Command Line

3.1 Jumping in the Deep End

At the beginning of this course, we said that we believe in active learning. Below are a bunch of exercises in roughly increaisng order of difficulty. Here's a bunch of advanced command line info you might find useful, though Google is probably your best friend.

Behind the Scenes: *You might wonder, "Why are we being given tasks that we haven't been taught how to complete?" The answer: that's science. One of the most powerful skills in bioinformatics is simply being able to parse documentation to figure out what you need to do. Taking the general knowledge from last lesson and applying it to scenarios you've never seen before is very, very valuable!*

3.2 Assignment

Get file1.txt and file2.txt:

File1.txt:

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=1DozuwGq6nQRSf2Dx6VNXSv6Vha8OMCIM' -O file1.txt
```

File2.txt:

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=17kYqxphGrkCK30FQuOUDIXOUhiZvjAMd' -O file2.txt
```

1. file1.txt contains the text of a Sherlock Holmes book. How many lines of the file does it take to recount his tale in Bohemia?

Does this command give you an idea of what you can search for?

```
head -n 70 file1.txt | tail -n 22
```

2. How many lines in file2.txt contain a period? (*Hint: The answer is not 60*)

3. Confirm your answers to 2-5 with an instructor or tutor. Then, delete the directory you downloaded in step 0.

3.3 Your Own File

So far, you've been working with files provided to you. However, for most things you'll want to do moving forward, you'll need to create your own. How do you do that?

3.3.1 vim

We're going to be using a text editor called "vim". Think of vim as Microsoft Word, but for your command line. You can use vim to open a file or create your own (just like Word!).

Here's a walkthrough of exactly what you'd need to hit on your keyboard to create a file called "new.txt" that contains "hello there" inside of it:

```
vim new.txt
i
hello there
'esc'
:x
```

1. `vim new.txt` - Open a file called new.txt in vim. If no file exists in the directory (true for us!) create a new one
2. `i` - Moves you into insert mode (where you're actually allowed to type)
3. `hello there` - Adds text into the file
4. `esc` - Moves out of insert mode
5. `:x` - Saves and quits the file

Use `ls` and `cat` to confirm that you've succeeded in creating this file.

3.3.2 Your Turn

Create an executable file called `greet.sh` that can be executed to print "Hello World" to the command line.

Tackle these subproblems:

1. What is an executable file?
2. How do I make a file print "Hello World"?
3. How can I execute a file?
4. [Optional, depending on if #3 fails] How can I make a file executable?

Update your program to print "Hello [your_username]". This should NOT be hardcoded, meaning if your program prints "Hello jsmith" on your laptop, it should print "Hello smansuri" on mine without any changes.

CHAPTER 4

Lesson 4: Python Basics

4.1 The Big Picture

So far, we've been working only on the command line. However, you'll often want to do more than the command line easily allows. Today, we're going to be learning the basics of one of the most popular languages for doing that: Python.

Side note: If you're familiar with Python, you many know there are two commonly-used versions, Python2 and Python3. Today, you will be using Python3, since Python2 is being deprecated at the end of this calendar year. They're very similar, so the skills you learn today should translate to any Python program you come across.

4.2 Getting Started

Create a new directory called "pydir". Enter the directory.

Python might not be installed on your workstations. In order to check: Type the following on the command line:

```
python3
```

You should see something that looks like this:

```
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type exit() or quit(), and move on.

4.3 How to Python

4.3.1 Hello World!

We're going to start by taking a quick look at how Python code is written. Create a file called Hello.py. The ".py" extension signals that you will be writing in python. Inside the file put:

```
print("Hello World")
```

Great! Now save + close the file, and run your newly-written program by typing the following on the command line:

```
python3 Hello.py
```

Because the "print" statement in Python outputs whatever follows it to the command line, you'll see your program print "Hello World". That was pretty trivial... let's try something more interesting.

4.3.2 Basic syntax

You'll need to obtain a file

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=1MOVGIAwde3Op4kApbn3Hw_C8_WETFt3Y' -O syntax.py
```

Now, open the syntax.py file and take a look inside.

Make sure you understand what's happening. Follow the comments closely, and ask one of us if you have any questions.

Now, run the code the same way we ran "Hello.py". You should see "Bioinformatics is Cool". Can you edit line 14 to make the program print: "is Bioinformatics Cool"?

vim-hint: navigate to a specific line in vim by going into command mode and typing `:14`

At this point, you can remove the "#" from the start of the last line (this is called uncommenting). Your Python senses should tell you that this line will now print out *myString*. Take a look at how *myString* is defined above and take a guess about what should be printed when you run the program. Once you're ready, run the program.

What if you only wanted to print *part* of your string, not the whole thing? Remember that a string is like a list (with the first character at index 0). So, what if we wanted to print just "Hello"? We can use specify a range of indexes to print from like so:

```
print(myString[0:5])
```

Note that the first number is the position of the first character printed (0 = 'H'), while the second number is **PAST** the last character printed (5 = ' ', but we only print up to index 4).

4.4 Indentation

Indentation in Python **matters**. Try adding a second print statement your "Hello.py" file so it looks like this:

```
print("Hello World")
    print("Indented line")
```

Now, try to run "Hello.py". Python will complain that there's a problem with your indentation (there was no need to indent, but you did anyways). You'll learn more about when to indent in the next section. Speaking of which, it's about time for some bioinformatics.

4.5 Loop-D-Loop

Say I have a fasta file containing genetic information (what is a fasta file?). Inside are some number of reads (remember reads from our *last lesson*?). I want to write a Python program that ONLY outputs the header lines (the ones that start with ">"). How can I do it?

```
-check every line-
  -if it starts with a ">"-
      -print the line-
```

This is one simple **representation** of how you could achieve this task. The **implementation** in Python, as we shall see, uses a loop. Which one of the three pseudocode lines above suggests we will need a loop?

Let's get a sample fasta file. Use the following command to download it straight in your working directory:

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=17NqX2e5jA9Jko-gV9lD5Bbnj9m2beKCJ' -O gencode.vM17.IncRNA_subsampled.fasta
```

Check the contents of your directory. You should see a file called "test.fasta".

Let's make a Python program that reads from this file. Create a new file called "Loop.py" and add this as the first line:

```
file = open("test.fasta", "r")
```

This will open the file (test.fasta) for "r"eading, and give you access to test.fasta in a variable called "file". Now let's use a loop to look at every line in the file:

```
file = open("test.fasta", "r")

for line in file:
```

That last line is the syntax for starting a for loop in Python. Next, looking back at our pseudocode, we see that we need to check if a line starts with a ">". Luckily, lines in a file are stored as strings! Remember that strings are indexed, meaning individual characters from them can be extracted using brackets (you might remember we did something similar above with!).

```
file = open("test.fasta", "r")

for line in file:
  if line[0] == ">":
```

Pay close attention to the indentation here. **You can think of everything that's indented after the `for` as being "inside" of the for loop** (it looks a bit like that too!). In our example, that means the `if` code is executed for every line in the file.

Finally, we just specify that we want the line printed if the line does start with ">". We indent the next line so Python knows it's part of the "if" statement, and...

```
file = open("test.fasta", "r")

for line in file:
```

(continues on next page)

(continued from previous page)

```
  if line[0] == ">":
    print(line)
```

Run Loop.py and see what happens. Voila.

4.6 Your Turn

Modify Loop.py so that it counts the number of G's and C's in the DNA sequences. Be sure not to count the header files. This value has real biological significance that you can read about here.

CHAPTER 5

Lesson 5: Bioinformatics x Native Python

5.1 Welcome

Congratulations! After your hard work honing your Unix and Python skills, you've been selected for a bioinformatic research project! Soon, you're going to be trusted with genetic data from an unknown source. Your goals will be to use bioinformatics to 1) analyze features of the data and 2) determine where this data came from.

5.2 Warm Up

Before you're allowed to work with real genetic data, you'll need be familiar with *command line* and *Python* basics.

Review these materials before moving forward, especially creating, editing, and writing Python3 files. Additionally, create and enter a `week3` directory to contain all the work you'll do today

5.3 The First Glimpse

5.3.1 Access

Great! You've now been granted access to the data you're tasked with identifying. Copy it to your working directory:

```
cp ~/../smansuri/unidentified.unknown .
```

As you've always done, explore the data to get a sense of what you're working with. (*This is a good check to see if you're understanding what to do. What are some techniques we've used in the past?*).

5.3.2 File Extenstion

The file you downloaded is called `unidentified.unknown`. Having worked with genetic data files before, you know what type of file this is. Change the file extension from `.unknown` to the appropriate extension. *For example,*

if you thought this was a PDF file, you would rename the file to `unidentified.pdf`.

5.4 Transcription Simulation

5.4.1 Complement

It's clear that this file contains a DNA sequence. Your colleague reminds you that DNA is double stranded and runs anti-parallel.

This means that a sequence "ACGT" from 5' to 3' (what's this?) is paired up with another sequence "TGCA" from 3' to 5'. These two sequences are called **complements** of one another. You can read more about DNA complementarity here.

He/She recommends that you also extract the complement of the unidentified sequence. Run the following on the command line:

```
cp ~/../smansuri/comp.py .
```

This will add an incomplete Python program called `comp.py`. Follow the instructions to determine the complement.

Once you get the program running correctly, you'll see a success message and a file called `unidentified_complement.fasta` created.

5.4.2 RNA Transcription

RNA is created by taking the template strand (in our example, the complement strand), and using an RNA polymerase to create a complement of the complement, but with Uracil (U) instead of Thymine (T). This means the RNA will have the same sequence as the original strand, but with U's instead of T's. For example:

```
Original:   ACTG
Complement: TGAC
RNA:        ACUG
```

Run the following on the command line:

```
cp ~/../smansuri/transcribe.py .
```

This will add an incomplete Python program called `transcribe.py`. Follow the instructions to transcribe the complement to RNA (*Hint: This will be very similar to your implementation in comp.py*).

Once you get the program running correctly, you'll see a success message and a file called `unidentified_rna.fasta` created.

Note: The template strand is read from 5' to 3', which is the orientation of our complement data. Therefore, you don't need to do any sort of reversal. You'll soon see another instance where you will need to reverse.

5.5 A Potential Breakthrough

You compare this RNA sequence with some others in your lab's database. The transcribed RNA you submitted has a sequence that's fairly similar to one seen before in rats.

There are two markers that could suggest your unknown sample may, in fact, be related to the one seen in rats:

1. The sequence "ATGGAGCTGACTGTGGAGGCATG" is often present.

2. The GC Content is above 55%.

Determine if this sequence appears in your sample, and see if the GC content is in the range you expect.

5.6 BLAST

Your colleague notices what you've been trying to do. She suggests you use an online tool called NCBI BLAST to compare your sequence to a database of sequences online. Click the link and use the Nucleotide Blast tool to copy-paste the **original DNA sequence** to determine what this sample is.

Great! You've identified what organism this comes from. What exactly is this organism? Is there anything particularly interesting about it?

5.7 Let's Try That Again

What you just did–exploring an unknown sequence–is one of the simplest and most common bioinformatic tasks. Ask yourself this: does everyone who wants to do this have to write their own algorithms, manually parse through sequences, and copy-paste into a web browser?

Nope. Let's explore a far more efficient method in the next lesson.

CHAPTER 6

Lesson 7: Data analysis with python

The power of Python partially lies in the many library it offers. For this lesson and as a general paractice for data analysis with python, go ahead and perform these imports:

```
#imports
import numpy as np
import pandas as pd
import scipy.stats as ss
from scipy.stats import norm
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
```

We have used lists in Python, but instead of using those built-in structures it is more common a practice to use arrays from the **numpy** library (**np.ndarray**) and series from the **pandas** library (**pandas.series** or **pandas.dataframe**).

As you can see form this table, it would make sense to use NumPy Arrays for calculations like taking average, sum, or even logarithm. Similarly, Pandas DataFrames allow tabular data access by 'index' or column headers and are hence ideal for reading and storing tabular data like excel files and CSV (comma-separated values) files. And the built-in list and tuple structures did give us flexibilty of having elements of differetn data types, which we lose in numpy and pandas array-like structures. All have their pros and cons, and are all useful in different scenarios. However, interestingly, they can all be transformed into each other by different functions as shown:

6.1 1. Getting familiar with numpy arrays

Let's start making numpy arrays

1. Using an array-like structure input:

```
np.array([[2,3,4,5], (2,3,4,5)]) # note that it can take list and tuple
```

1. Using a range of values:

```
np.arange(5)
```

(5 is stop value, check docs for start and step)

1. Using random values:

```
    np.random.seed(3) # setting a random seed
    np.random.randn(12) # 12 random values (from a standardised normal distribution)
```

A numpy array x can be visualized as n-dimensional matrix with shape given by x.shape, total number of elements by x.size, transpose by x.T and number of dimensions (n) by x.ndim

Example:

```
x = np.array([[2,3,4,5], (2,3,4,5)])

print(x.size) #8
print("Before transpose, shape: ",x.shape) #(2,4)
y = x.T
print("After transpose, shape: ",y.shape) #(4,2)
```

Using np.sum(x) we get (you guessed it) sum of all values in x. But using axis pararmeter of same function, we can get array of sum of each column of x.

```
print(np.sum(x, axis =0)) # [ 4  6  8 10]
print(np.sum(x, axis =1)) # [ 14 14],       axis = 1 means sum the columns
# print(np.average(x, axis =1)) gives you average
```

Adding 2 Numpy arrays will give the result of additions element-by-element

```
a = np.array([1,2,3])
b = np.array([3,-1,0])
print(a+b)
```

Values can be accessed just like lists:

```
print(x[1,3]) # 5 (value in 1st row, 3rd column
print(x[1][3]) # same as above
print(x[:,1:3]) # [[3 4], [3 4]] (values in all rows, from columns of index 1 to 2
```

np has function that can be applied on all elements

```
print(np.arcsin(I))
```

a*b is element-by-element product of a and b np.dot(a,b) gives dot product of a and b

```
print(I*[[2,1],[1,5]])
print(np.dot( I, [[2,1],[1,5]] ))
```

You can concatenate or stack vectors vertically (np.vstack()) or horizontally (np.hstack())

```
a = np.arange(5)
b = np.vstack([a, [9,2,0,9,3] ])
print(a)
print(b)
```

We can also perform element-wise operations on arrays of higher dimensions using that of lesser dimensions or even scalars. The effect of the operation simply gets applied throughout the 'extra dimension'. This is called **broadcasting**.

```
print("a+3:",a + 3) # adds 3 (scalar) to all elements of a
print("b + 100: \n",b + 100) # adds 100 (scalar) to all elements of b
print("b * [[3],[100]]: \n",b * [[3],[100]]) # all values in first row get multiplied␣
↪by 3, 2nd row by 100
```

Numpy arrays are great for all sorts of calculation-based analysis and will be used frequently.

6.2 2. Using pandas

Pandas is a friendly way to read, analyze, and write numerical datasets of a variety of formats. We will use CSV (Comma-separated Values) which uses a comma as a delimiter. We explore two examples - a small dataset, and a large dataset

6.2.1 2.1 Small Dataset example: Ladder Distances (Read, Plot, Regress, Write)

In gel electrophoresis, a DNA band travels farther if it has greater number of base pairs. We have data for band distances (cm)

There is a non-linear direct relationship between band distances and length of fragments, but the constant depends on the gel properties. We use a standard library of known fragment sizes that appears like 'ladder' and measure the distances from the wells of the 'rungs' or the bands. In Fig 2, the ladder is λHindIII. Then by measuring distance from the well for our samples (A(1,2,3),B(1,2,3)), we can determine the corresponding length of DNA fragments.

Let's get Ishaan's BIMM 101 (Recombinant DNA lab) ladder distance readings:

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=1QDnwIniSqER03VxS5jGNdBLq4dZ1t66o' -O ladder_distances.csv
head ladder_distances.csv
```

Now let's form a pandas.DataFrame object 'ladder_dist' containing this data

```
ladder_dist = pd.read_csv("./ladder_distances.csv")
"""Data types"""
# print(ladder_dist.dtypes)
"""Object type :"""
print(type(ladder_dist))
```

Notice that pandas assigned the int and float types automatically. Pretty cool, huh?

Let's first get the column names using `ladder_dist.columns`

- The columns headers are the first line of our file
- This is not an absolute requirements for all csv files
- Check documentation for pd.read_csv() and check parameter 'header'

Now let's use column names as inidces to get particular columns

```
bp = ladder_dist["bp"]
dist = ladder_dist["distance"]
```

Use ladder_dist.index to get/set row indices

```
# Get row indices (currently range of numbers 0 - 7)
print(ladder_dist.index)
# Row indices can also be set
ladder_dist.index = ["first","second","third","fourth","fifth","sixth","seventh",
↪"eighth"]
ladder_dist
```

Access columns/ rows/ sections from data

```
# Get a row based on label
ladder_dist.loc["second"]
# TODO: check type (HINT: it's Series)

# Get a row based on int index
ladder_dist.iloc[0]
# TODO: Use iloc to get the first 3 rows (notice now DF is returned)

# Get a column by label as Series
# ladder_dist["bp"]
# or
ladder_dist.bp

# Get a column by label as Dataframe
ladder_dist[["bp"]] # (looks lit to me)

# Get a column by int index
ladder_dist[ladder_dist.columns[0]]
ladder_dist[[ladder_dist.columns[0]]]
```

Now feel free to read pandas documentation and get particular cells:

```
#Get particular cell
# By label
# TODO: Get the distance of the fifth gel band using ladder_dist.at

# By int index
# TODO: Get the same using ladder_dist.iat
```

2.1.2 Visualize the data

Let us visualize some data. Read **matlabplotlib.pyplot** documentation, and get a dot plot of 'bp' versus 'dist'. (What should be on the x-axis?)

Does this look like a straight line to you? How about the log of this line? Now dot plot log(bp) versus distance. And kidly do this in the same python file or same ipython notebook cell.

You might realize that to plot them separately, you need something. And that something is `plt.show()`. It plots all *open figures* above it.

Also note there might be multiple functions in pyplot to achieve the same task.

Another way to visually represent the linear relationship between log(bp) and dist is to draw semi-log standard curve: i.e graph bp-dist, but set the x-axis log-scale.

```
# Log-scale plot
plt.xscale('log') # or plt.gca().set_xscale("log")
# TODO: now plot bp versus dist
```

2.1.3 Bring in Statistics

How strongly are log(bp) and diatnce related? Or should we ask,

How strongly are log(bp) and diatnce correlated?

There are different types of correlation coefficients for simliar purposes. Two of them are

- **Spearman Rank Correlation Coeffiecient**: Describes monotonicity
- **Pearson Correlation Coeffiecient**: Describes linear relationship

Which correlation method do you think is more appropriate for the relationship between bp and distance?

Which correlation method do you think is more appropriate for the relationship between bp and distance?

Let's answer these questions by using **scipy.stats** (imoprted as ss) functions to actually get both pearson R coefficient and p-value and Spearman R coefficient and p-value for

- bp versus distance
- log(bp) versus distance

Using the r and p-values,can you now answer the more appropriate method questions better?

While we had observed a strong linear relationship between log(bp) and dist from the dot plot, we now have a statistical measure backing it up.

What we can now do is

- derive the equation of that best-fit line
- Use that equation to get bp in our samples for known distances

So let's first get thhat line of best-fit. This is a type of machine learningn called **Linear Regression** and thus we start our machine learning component.

For machine learning, one of the most popular python packages is **sklearn**.

So let's start machine learning! Since we're performing linear regression, we build a linear regression model object `linregressor` using `linear_model.LinearRegression()` from `linear_model` that we already imported from `sklearn`.

Now `sklearn.linear_model.LinearRegression()` models take parameter X and Y, where X is the array containing each data point's 'feature vector' (a vector containing features that describe the input part of the data point). In our case, the feature vector for each of the eight points contains just one feature - the dist if that data point. So we just need to transpose dist, a row vector, and use the column vector as X, and similarily use the transpose of bp as Y. We know that transpose of np.array `a` is `a.T` (Hurray!) However, for 1-dimensional numpy array `a`, `a.T` returns the same row vector `a`, so it's not that straightforward. (boo!)

One simple way to get our Y and X is to:

```
distX = [[x] for x in dist]
log_bp = [[y] for y in np.log(bp)]
distX,log_bp
```

You are really getting the hang of it if you came up with that, but also numpy has a special function to convert 1-D to 2-D `np.atleast_2d()`. (Hurray!)

So to get the transpose of `a`, just use `np.atleast_2d(a).T`.

```
# feel free to try taking transpose a 2-dimensional vector
a = np.arange(5)
print(a)
```

(continues on next page)

(continued from previous page)

```
print(np.atleast_2d(a))
print(np.atleast_2d(a).T)
```

Now that we have column vectors `distX` and `log_bp`, lets fit the model with `linregressor.fit(distX, log_bp) # fit([x values],[y values])`

Now we can get the parameters *m* and *c* using `linregressor.coef_` and `linregressor.intercept_` respectively. Can we use these parameters to get our best-fit line?

Plot the best-fit line continous (not dot) for our *x* (dist) values with the actual *(x,y)* (i.e (dist,bp)) data points as dots. Use the code given below.

```
plt.plot(dist,bp,'o')
plt.plot(distX, np.exp(linregressor.coef_*distX + linregressor.intercept_,))
# or to get the line starting from x = 0
# plt.plot( np.hstack([[0],dist]), np.exp(linregressor.coef_*np.hstack([[0],dist]) +␣
↪linregressor.intercept_)[0] )
plt.yscale('log')
plt.xlabel("Distance traveled by the band (mm)")
plt.ylabel("Length of DNA fragments in the band (bp)")
```

You should be getting a graph like Fig 3.

This line of best-fit lets us predict the lengths of fragments for sample A (dist = 18 mm) and B (dist = 26.16 mm). Besides getting `linregressor.coef_*dist_A + linregressor.intercept_` to get the log(bp_A), we can also use `linregressor.predict(dist_A)`.

```
print(np.exp(linregressor.coef_*[18,26.16] + linregressor.intercept_))
print(np.exp(linregressor.predict( [[18],[26.16]] )))
```

Ishaan's hand-drawn semi-log curve gave the answers 3900 bp and 1000 bp. Do your answers seem consistent? Feel free to use the plot to convince yourself.

So shall we move to move to the large dataset? This dataset is from a historical paper *"Molecular classification of cancer: class discovery and class prediction by gene expression monitoring"* and we will use that for our classification workshop in the next lesson :)

CHAPTER 7

Lesson 8: Machine Learning - Classification, Dimesionality Reduction

You have already started Machine Learning when you performed the linear regression analysis, but let's talk about Machine Learning in general first, then, as promised earlier, we'll move to our larger dataset.

7.1 Supervised Learning:

Use training set with correct inputs and outputs to predict outputs for test data inputs.

7.1.1 Classification:

- Inputs(X): Features
- Outputs(y): binary or multiple classes

7.1.2 Regression:

- Inputs(X): Independent Variable
- Outputs(y): Dependent Variable (Continuous)

7.2 Unupervised Learning:

Find patterns among inputs (features), no labels in data

7.2.1 Clustering:

- Find groups within data (Example: Phylogeny tree)

7.2.2 Dimensionality Reduction:

- Find a lower dimension representation of higher dimensional data

We built a linear regression model in the last lesson, and Classification and Dimensionaloty Reduction component of the ML lesson is currently available as Kaggle Notebook on Tumor Classification between AML and ALL and finding top genes contributing to the classification.

Next, we will explore Clustering.

CHAPTER 8

Lesson 9: Clustering

Syntax note:

Note that in Python syntax, you will have to use a dot to specify where you want to get a function from. For example, there is a package called numpy, with a module called random, with a function called seed. The seed function is called by typing out `np.random.seed()`, telling python where exactly it needs to look.

WE ARE USING PYTHON3, NOT PYTHON

8.1 General Factoids

Clustering: The process of partitioning a dataset into groups based on shared unkown characteristics. In other words, we are looking for patterns in data where we do not necessarily know the patterns to look for ahead of time.

In: N points, usually in the form of a matrix(each row is a point).

In: A distance function to tell us how similar two points are. The most intuitive distance function is euclidean distance. The type of distance measure you use can vary depending on the type of data you are using. One specific example of how fine tuned distance metrics can be is a distance metric that weights substitutions in DNA strings heavier than indels because indels are more common sequencing errors. This means that such a metric would not separate two DNA strands due to sequencing error.

Out: K groups, each containing points which are similar to each other in some way. Some algorithms cluster for a preset number K, others figure it out as they go along.

How is DNA a point? One common way we translate DNA strings to points is by making a string of DNA into a kmer vector. A kmer is a string of length k, and there are 4^k possible kmers in any DNA strand. Usually we think of kmers as ordered lexicographically like this: AAA, AAC, AAG, AAT, ACA, ACC, ACG, ACT, AGA and so on. One way to represent a DNA strand is to create an array of zeros of length 4^k and increment by one for each time that kmer appears in the DNA strand.

Problem: Find the kmer vector of "AG" (k=2).

8.1.1 Bioinformatics Applications of Clustering:

1. Finding what genes are up and down regulated under certain conditions. Imagine you have a matrix, where each point is a set of gene expression recorded for a variety of conditions. If two points are close to each other, that means they had similar expression levels throughout those conditions.
2. Discerning different species present in a sample of unkown contents. This can be done with an algorithm that does not have a predetermined amount of clusters. An example application is taking HIV sequences from a patient, clustering them, and filtering clusters under a certain size to find the sequences of prevalent strains within the patient.
3. Finding evolutionary relationships between samples using hierarchical clustering. The earlier on two centers were combined, the closer their corresponding points are from an evolutionary perspective.

8.2 Generating Data

In order to demonstrate the differences between different clustering methods, we need datasets that cluster differently depending on technique.

Copy the starter file into your directory:

`https://drive.google.com/file/d/13j7v4lUOfWY4PhR9mT1-SBao8FkitbEh/view?usp=sharing`

This file `clust.py` contains a plotting function - don't touch it! We will be creating stuff for that plotting function to plot in a minute, so write your code below that function.

Start by creating some blobs using numpy's random normal generator. First, create a seed for the random number generator with

`np.random.seed()`

Make a few blobs of points (probably two) with centers between 0 and 20 and varied stdevs (keeping the array dimensions the same). Example syntax: `clust1 = np.random.normal(5, 2, (1000,2))`

Put the point blobs into one structure so that we can cluster them all at once with

`dataset1 = np.concatenate()`

We will be comparing how our point blobs cluster to the way that circles of datapoints cluster. Create two concentric circles as the second dataset. `dataset2 = datasets.make_circles(n_samples=1000, factor=.5, noise=.05)[0]`

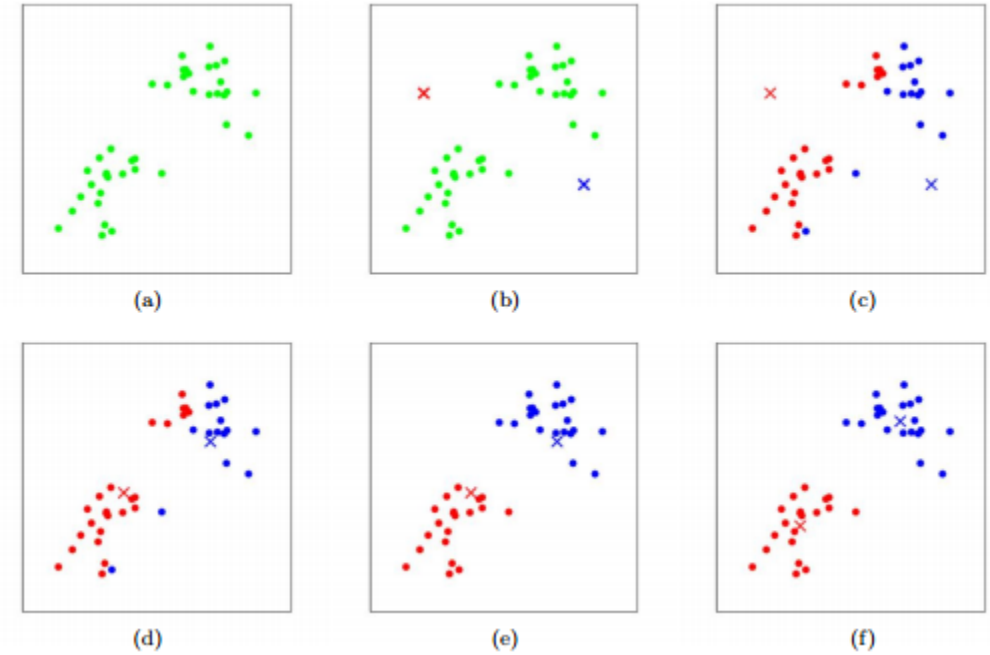
If you want to see what your datasets look like, use the `cluster_plots` function

Note: `cluster_plots` will save your plots to a pdf file in your current directory, the name of which can be changed.

```
scp username@ec2-3-135-188-28.us-east-2.compute.amazonaws.com:/home/username/path /
↪localpath/
```

8.3 Clustering Data

8.3.1 1. K-means



Steps:

(We present the **Lloyd Algorithm** here, but there are other algorithms for k-means clustering) I. Select K centers. This can be done a variety of ways, the easiest of which is to select *k* random points; maybe find a random point, find the farthest point from that and select it, find the furthest point from the previous and so on; this is called farthest first traversal

II. Assign each point to the center nearest to it. *(Centers to Clusters step)*

III. For each center, take all the points attached to it and take their average to create a new center for that cluster.

IV. Reassign all points to their nearest center *(Clusters to Centers step)* and continue reassigning + averaging until the iteration when nothing changes.

Code:

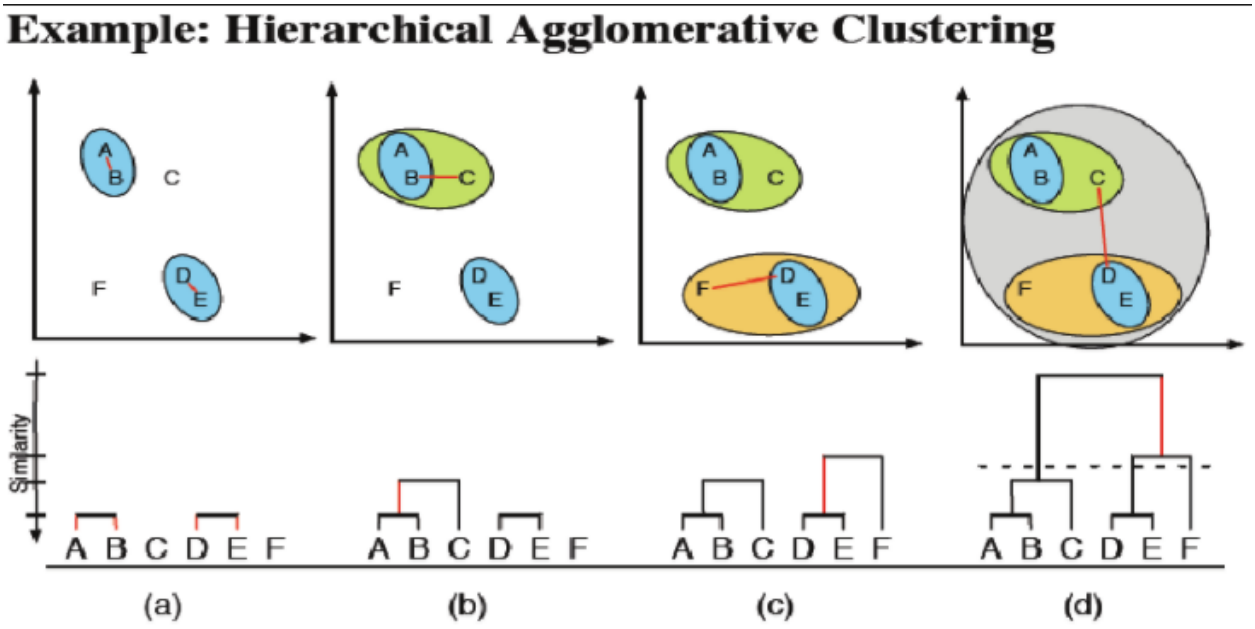
We will be using the KMeans algorithm from sklearn's cluster package to dataset1 and dataset2. Remember that k is the number of clusters the KMeans algorithm will assign points to, so give Kmeans the appropriate k. The points returned from the KMeans function will be separated into clusters, making them easy for the cluster_plots function to distinguish and color accordingly.

```
kmeans_dataset1 = cluster.KMeans(n_clusters=k).fit_predict(dataset1)
cluster_plots(dataset1, dataset2, kmeans_dataset1, kmeans_dataset2)
```

(While using the sklearn implementation as a black box is the general practice as we've seen so far, it is a good learning practice and a nice coding challenge to implement the Lloyd Algortihm yourself. Check out this Rosalind problem

"Implement the Lloyd Algorithm for k-Means Clustering" (You could solve the previous problem on Rosalind for distance calculation)

8.3.2 2. Agglomerative Hierarchical



Description:

I. Start with every point being its very own cluster center.

II. Find the two centers which are closest to each other and combine them into one cluster. The new center is the average of the two previous ones.

III. Continue combining the closest centers until all points are under a single cluster.

Code:

Let's look at the blobs and circles we made again. Look at the documentation page for agglomerative clustering and figure out the proper calls for the two datasets (syntax is very similar to what you did for k-means).

Now, run `cluster_plots()` to graph and `scp` to view the results. Do you notice a difference in quality of the clustering of the left and right graphs?

Unfortunately, this still does not solve our circlular cluster issue. For that, we can ask the sklearn package to build a graph out of our circles which restricts the amount of nearest neighbors a point can cluster with.

```
#this line limits the number of nearest neighbors to 6
connect = kneighbors_graph(dataset2, include_self=False, n_neighbors = 6)

#in this line, you need to set linkage to complete, number of clusters to 2, and set␣
↪connectivity equal to the graph
#on the previous line
hc_dataset2_connectivity = cluster.AgglomerativeClustering().fit_predict(dataset2)
```

Use `cluster_plots()` to graph the plot resulting from the regular AgglomerativeClustering beside the graph that plots the connected AgglomerativeClustering. As you may have guessed, this fixes the circle problem.

8.3.3 3. Soft Clustering

Description:

This is essentially the same thing as k-means, but points cannot be hard assigned to a cluster. Instead, each point has a probability of being in each cluster, and points with higher probabilities influence the center of that cluster more.

I. Select K centers randomly (or slightly less randomly, like farthest first traversal.

II. Expectation step: Assign a responsibility (a likelihood) for each point in respect to each cluster. Basically, the closer a point is to a center the higher the likelyhood of that point. This can be calculated in a variety of ways, but the main idea is that it is some function relating the distance from a point to a center to the distances between all points and that center to see if it is much closer or further than other points

III. Maximization step: compute new cluster centers by using a **weighted** average of the points based on the likelihoods calculated in step II.

Code:

Let's go ahead and see what will happen with our blobs and circles under this clustering algorithm:

```
em_set1=mixture.GaussianMixture().fit_predict()
```

Use these new clusters in our `cluster_plots()`

Also feel freee to check out Rosalind problem "Implement the Soft k-Means Clustering Algorithm"](http://rosalind.info/problems/ba8d/).

8.4 Challenge

Head over here to complete our clustering challenge!

8.4.1 Credits

Lloyd Algorithm k-means clustering and Soft Clustering explanation and challenge are adapted from Rosalind, particularly Rosalind problem "Implement the Lloyd Algorithm for k-Means Clustering" and Rosalind problem "Implement the Soft k-Means Clustering Algorithm"

CHAPTER 9

Dirichlet Process Means Clustering Challenge

Copy the starter file from `/home/ubuntu/clustering_lesson/dmp.py`

The dirichlet process is a method of clustering without knowing the number of clusters ahead of time. The means part of the name indicates that we will be defining clusters by taking the mean of the members in the cluster. Something like this algorithm may be used for example 2 in the list of clustering related bioinformatics problems. Here are the steps:

I. Add the first point in your data as the first center. A single data point is a numpy array.

II. Iterate through all of the points in your data, calculating the distance between the current point and all existing centers.

A. If the point falls within a certain threshold distance of the center, add that point to that center's cluster

B. If the point does not fall within a certain threshold distance of the center, add that point to the list of centers

III. Once you have gone through the Dirichlet Process, you do the means part. Redefine the centers of each cluster to be the average of all points in the cluster.

IV. Repeat steps II and III either until an iteration provides no change in the assignment of points to centers, or a maximum number of cycles is reached.

HINT 1: You will need an array of arrays to keep track of which points are in which clusters. Array 1 will contain the indices of the points in the first cluster and so on. If a point moves from its current cluster, remember to take it out of the old cluster before adding it to its new cluster.

HINT 2: You will need an array of arrays to contain the centers of each cluster. The first array will be the center of the first cluster and so on.

CHAPTER 10

Lesson 10: Alignment

10.1 Alignment Basics

Sequence alignment is the process of putting similar regions closer to each other, and inserting gaps (denoted by '-') where there may have been an insertion/deletion event. Identifying regions of similarity can help us identify structural, functional, or evolutionary relationships between sequences.

There are hundreds of alignment methods for different types of information (DNA vs amino acid), different uses (finding common domans vs comparing homologous genes), and different special cases (antibody sequences, viral sequences). If you are interested in learning details about how some existing methods work (this is BENG 181 material): the most generic DNA alignment method is a pairwise dynamic programming method called Smith-Waterman, alignment of many sequences (multiple sequence alignment) can be done by Fast Fourier Transforms, and alignment for accurate amino acid sequence homology is done by HMM profile alignment.

This lesson will focus on hands-on learning of three types of alignment: global, local, and multiple sequence alignments.

10.2 Pairwise Alignemnt

Pairwise alignment is the alignment of one sequence to one other sequence. In order to look in detail at the usefulness of different types of alignments, we will start by looking at pairwise alignment methods implemented with the Smith-Waterman method.

10.2.1 Global Alignment

In this case, the word "global" just means that the **entire** first string is aligned as best as possible to the **entire** second string. It's used to compare two homologous sequences, like comparing the same gene between individuals or species.

Make a new Python file `aligners.py` and import some stuff:

```
# Import pairwise2 module
from Bio import pairwise2

# Import seq objects
from Bio.Seq import Seq

# Import method for formatting alignments
from Bio.pairwise2 import format_alignment
```

Define strings `TGCCTTAG` and `TGCTTGC` and using global alignment on them:

```
pairwise2.align.globalxx()
```

The alignment method returns a list of the most high scoring(good) alignments. You can print those out by iterating through the alignments with a for-each loop and print them out one by one. Use this syntax to make them look better during printing:

```
print(format_alignment(*alignment_name))
```

What is that asteriks? In different languages an asteriks has different meanings, but in Python it denotes a variable quantity of arguments. In other words, methods that want this asteriks are capable of accepting either one, maybe two, or maybe more arguments(inputs).

10.2.2 What exactly is a good score? Why is one alignment better than another?

Glad you asked! If you look at the score printed out below each alignment, you will notice that the score is coincidentially identical to the number of nucleotides that match between the two aligned sequences. The alignment we tried gives gaps, insertions, and deletions a score of zero and matches a score of one. Other alignments may have more complex behaviors, including negative scores for insertions/deletions, custom scores based on which nucleotide is mismatched with which, and more. There are several modes of alignment available on BioPython, each with customizable scoring. Look over here if you want to see the range of what BioPython has to offer.

Take the two strings `ATGCGGCCATTATAAGTGGTCTCT` and `GATTATAATT`, for instance. In case you want to reward +10 for matches, -4 for mismatches, you could use: `pairwise2.align.globalms(str1, str2, 10, -4)` Now suppose you add gap penalty, deduct 3 for opening a gap (an indel not preceded by an indel), and deduct 2 for extending a gap (an indel preceded by an indel). `pairwise2.align.globalms(str1, str2, 10, -4, -3, -2)` For now, let us use +1 for matches, -1 for mismatches, -1 for opening a gap, and -1 for extending a gap.

10.2.3 Local Alignment

While the Global alignment method above gave us all the highest score alignments, we might not want any of them. Do you think it is fair for strings `ATGCGGCCATTATAAGTGGTCTCT` and `GATTATAATT` to be aligned from the begining of both strings to the end of both strings? Notice that starting at index 7, `"CATTATAAGT"` in the first string looks more similar to the second string than any global alignment if we ignore the indel penalties at the flanking ends of the second string aligned and could have been a good place to find a conserved region.

The word "local" means the best alignment between two **subsequences**. This method can be used when looking for a conserved gene between two otherwise very different organisms. Aligning the messy differences between two different species is not useful, but finding two subsequences (two genes) that the species have in common without aligning the whole sequences can be very useful.

Now, let's compare how these strings align with globally vs locally. (Yes, there is a `pairwise2.align.localms()` function in biopython module)

Looking at the matching nucleotides in global and local alignments of these string, which one makes more sense? Does it make sense to care about the matches the global alignment has at the very end of the sequences?

10.3 Multiple Sequence alignment

Multiple sequence alignment aligns multiple sequences, but its inner workings are a bit complicated (my way of saying I do not know them well enough to teach them) so we are just going to focus on their applications. This type of alignment is used on a large number of more or less related sequences in order to infer homology and build evolutionary trees. My multiple sequence aligner of choice is mafft, which can be called from inside python.

Grab `D_db.fasta` and `J_db.fasta`

```
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1qHc1__
↪7e5em8z4AS9D7hygCFNnIo8MkW' -O D_db.fasta
wget --no-check-certificate 'https://docs.google.com/uc?export=download&
↪id=1wz7S4CgbhHBujx134MCjKahMAKPuPslr' -O J_db.fasta
```

Now align them with the default settings on mafft:

```
import subprocess

in_file = "D_db.fasta"
subprocess.call(["mafft", "--out", "aligned.fasta", in_file])
```

D_db.fasta and J_db.fasta are the databases of known human diversity and joining genes, respectively. These genes are building blocks of the heavy chain on an antibody. We can see that the J's align pretty neatly, with some deletions, while the D alignment is a mess - many mismatches and gaps. This clearly indicates that J genes are much more highly conserved than D genes, which you probably guessed from their names. Find answers to your questions about heavy chains over here

Now that we are familiar with what alignment has to offer, let us get hands-on with a specific implementation:

10.4 Codon Alignment

Challenge: Codon Alignment

This challenge describes a method of alignment that can be used to track the differences between sequences on the codon level. The goal is achieved with mutliple sequence and some basic sequence manipulation tricks.

CHAPTER 11

Codon Alignment Challenge

One thing biologists care a lot about is the way amino acids change through time. This is found by sequencing DNA at several timepoints, codon aligning various timepoints, and comparing timepoints to see which amino acids change at what time. Why don't we just convert to amino acids and align there? Because different reading frames and deletions cause suboptimal translations into the amino acid realm, and therefore decrease the usefulness of the alignment. Plus, codon alignment can reveal synonymous vs nonsynonymous mutations.

For a visualization of this type of alignment, head over here. The sequences shown here are from an HIV envelope. If you click on on amino acid index, you can see a graph showing how the amino acid in that position evolved over time. You can explore the site - I recommend the tree section because it is pretty.

I made a fake fasta of sequences that need to be codon aligned over at `/srv/WI20/not_aligned.fasta`. Here is my step by step guide on making your own codon aligner to make sure those sequences are nice and neat.

11.1 0. Get the imports

```
#to call mafft
import subprocess

#to use the Seq type
from Bio import Seq

#to read/write fasta
from Bio import SeqIO

#to translate
from Bio.Alphabet import generic_dna
```

11.2 1. Align with mafft and replace initial gaps with N's

11.2.1 Why we do this?

Due to DNA fragmentation, sequencing starts at many different points, and we don't know ahead of time where the points are. If we don't identify where one sequence starts relative to another, we cannot begin comparing them. Why the N's? That's because the biological meaning of N's is different than that of gaps. Gaps in an alignment indicate deletions or insertions from one sequence to another, we predict that something was removed or added. In the case of different starting points, we know that *something* is supposed to be there, since we have information from other sequences. Thus it is not an insertion or deletion, but unkown nucleotides. We represent this with N's.

a. Align the file with mafft. Syntax for this is provided in lesson 8

b. Go through each of the sequences in the mafft aligned file

```
for seq_record in SeqIO.parse("nuc_aligned.fasta", "fasta"):
```

c. Count the number of initial gaps on each sequence

d. Degap each sequence and place it into a Seq object

The syntax for this step is whack, so let me give it to you:

```
sequence=Seq.Seq(str(seq_record.seq).replace("-", ""))
```

d. Prepend each sequence with n's. The number of n's should be equal to the number of initial gaps that sequence had.

11.3 2. Correct the reading frame and length

We have now lined up sequences with different starting points, but what if every single sequence starts at the wrong reading frame? All of the translated animo acids will be useless. One way of making the choice between starting each sequence from the first, second, or third nucleotide by seeing which one results in the longest total distance between stop codons in all of the sequences.

In the interest of time, let us assume that the reading frame is correct. All you need to do is to make sure that the alignment has a length that is a multiple of three.

a. Go through each index from 0 to the number of sequences

b. Use the modulus to find whether the current sequences has a length divisible by three.

What is a modulus? it's this `%` symbol. In math, this symbol will find the remainder for you. So 15%3=0, 16%3=1, 17%3=2, 18%3=0 and so on.

c Based on the remainder you found in b, figure out how much of the sequence you should cut off the end. Syntax hint: `seq[0:-1]` will give you the sequence with the last nucleotide cut off.

11.4 3. Translate the DNA into amino acids.

How? Google translating DNA with BioPython.

11.5 4. Mafft align the amino acid sequences

It's the same syntax as step 1, just put your animo acids in a file.

11.6 5. Backtranslate to codons

We need to backtranslate from amino acids into codons by inserting the gaps from the amino acid alignment into the nucleotide codons themselves.

a. Go through each amino acid in the aligned amino acid array

b. If it's a regular amino acid, go ahead and take next three nucelotides from the degapped nucleotide array (the one with the n's). If it's a gap, think about how many gaps that corresponds to in nuceotide space (hint: it's three) and insert those gaps.

11.7 Answer Key

Verify your answers:

The code itself can be found at `/srv/WI20/codon_align.py`.

The answers for each question are currently being made. They will be located in `/srv/WI20` (soon).

CHAPTER 12

Lesson 11: Phylogenetics

12.1 The Basics

12.1.1 What is Phylogenetics?

At its core, phylogenetics is the study of finding evolutionary relationships between organisms. How, you ask? By looking for similarities (in our case, on the DNA level). Let's pose one such problem:

Imagine you're given five distinct organisms and are told to figure out how they're evolutionarily related (that is, how they evolved from one another). Two questions pop into your head: Why? and How?

12.1.2 Why Do I Care?

The answer is twofold:

1. On a broader scale, an understanding of phylogenetics gives us a better understanding of the living world, and how we + our fellow organisms fall into it. We can answer questions about how we are linked to jellyfish, and how that relationship has shaped the world around us.
2. On a smaller scale, phylogenetics also provides key information about genetic drift, the evolution of genes and diseases, and the development of molecular differences between organisms.

12.1.3 How Does it Work?

We need some shared, conserved mechanism of comparing the organisms… something like DNA! There are sequences in DNA that are highly conserved between organisms (usually sequences that code for absolutely essential parts of life) that are excellent candidates for comparison. For example, imagine you have three organisms with the following sequence of DNA in a conserved region:

```
1. ACGTG
2. ACGTC
3. ACGCC
```

We can compare the nucleotides at each position and find that 1 and 2 differ by one nucleotide, 2 and 3 differ by one nucleotide, but 1 and 3 differ by **two nucleotides**. This might lead you to conclude that the relationship between organisms 1 and 3 is the most evolutionarily distant.

Note: In reality, these conserved sequences are much, much longer, and the analysis is far more robust than counting of a few nucleotides.

12.2 Biopython

If you're not familiar with Biopython, please skim our previous lesson here.

If you know what you're doing, running this on your own machine instead of on EC2 will probably give you some nicer graphs. This is optional, continue on EC2 if you wish!

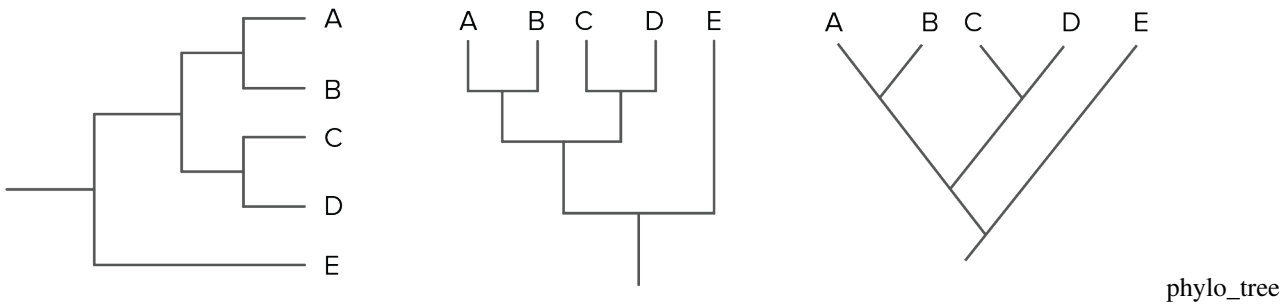
You'll need to have Biopython installed to continue (done for you on EC2). Confirm by running a Python program with the following:

```
from Bio import Phylo
```

If nothing shows up during execution, you're good to go!

12.3 Find the Relationships

12.3.1 Working with Trees



phylo_tree

You've probably seen one or more of the phylogenetic trees diagrammed above. These visually represent the heart and soul of phylogenetics. Because we use a **last universal common ancestor model** (LUCA model), in which the root represents an ancestral organism from which all living organisms today evolved, each evolutionary relationship can be represented by a branching of the tree.

12.3.2 The Data

Let's see this in action with our own phylogenetic tree! Remember those five organisms we mentioned earlier? Well here they are, each with a conserved DNA region sequenced:

```
   5   13
Alpha      AACGTGGCCACAT
Beta       AAGGTCGCCACAC
Gamma      CAGTTCGCCACAA
Delta      GAGATTTCCGCCT
Epsilon    GAGATCTCCGCCC
```

Note: The first line is a header. The first number is the number of organisms, and the second is the length of each DNA sequence.

Copy and paste all of the data above (including the header file) into a new file called `msa.phy`. The `.phy` extension indicates that the file contains phylogenetic data.

What kind of data goes into making trees?

As with basically all other programs dealing with biological sequences, phylogenetics algorithms need you to align your sequences before you input them, so that they are easier to compare. If you are curious about different methods of alignment and their influence on the final alignment go over to our lesson on alignment over here.

12.3.3 The Setup

Now we're ready to start analyzing this data. Create a new python file and add the following to import everything we need:

```
from Bio import Phylo
from Bio.Phylo.TreeConstruction import DistanceCalculator
from Bio.Phylo.TreeConstruction import DistanceTreeConstructor
from Bio import AlignIO
```

Make sure you can run this file without errors!

Here's some documentation that may help you if you get stuck: AlignIO, DistanceCalculator, DistanceTreeConstructor

Note: If you're not able to follow the instructions below, ask the bootcamp developers for a templated file to help guide you!

12.3.4 Alignment

The first step is to import the (already aligned) sequences into our program. **Use AlignIO to read in the data, and store it in a variable `aln`. Print `aln` to confirm this step worked correctly.**

12.3.5 Distance Matrix

Now that we have our data, we're interested in how similar (or different) the sequences are from each other. The more similar, the more likely they are to be evolutionarily close. Therefore, we use a distance matrix to store information on how similar each DNA sequence is from every other sequence. **Create a Distance Calculator object with the 'identity' model. Use it to calculate the distance matrix of the alignment, and store it in a variable `dm`.**

12.3.6 Phylogenetic Tree

Now, the real magic... creating the phylogenetic tree! We can use our Distance Tree Constructor to make a tree from our distance matrix. **Use the UPGMA algorithm in Distance Tree Constructor to create a tree and store it in a variable `tree`.**

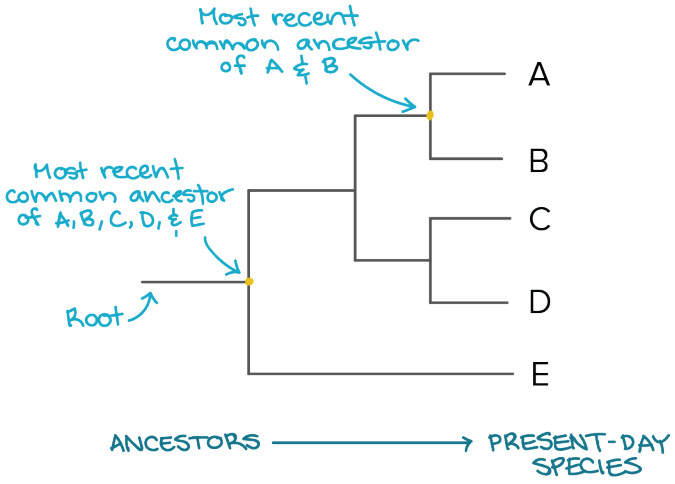
12.3.7 Visualization

Great, you're done! Except... you're probably interested in how your tree turned out! The simplest way to visualize your tree is to **use the `Phylo.draw_ascii()` method to print the tree to the terminal.** This will give you a low-detail sketch of your tree. *Make sure you can do this before moving forward.*

You're probably interested in a much more detailed tree, though. The `Phylo.draw()` method provides that, but requires `matplotlib` as a dependency (`pip install matplotlib` if you don't have it [EC2 does!]).

12.3.8 Interpretation

Phylogenetic trees are interpreted using the following terms:



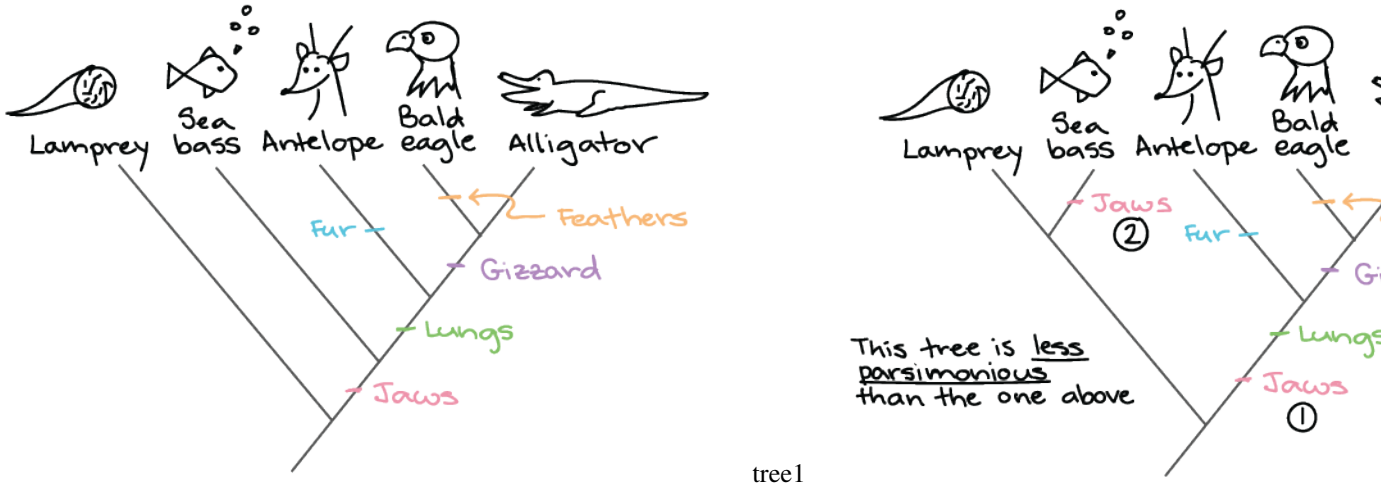
tree

How many common ancestors are there? What does that tell you about the relationships and evolutionary distances between the organisms.

12.4 Challenge: Different Algorithms, Different Results

We made two algorithm choices in our pipeline above. One is the distance calculator's model and the other is the distance tree's model. Use the documentation to find alternate algorithms and try various combinations. You'll see the predicted tree (and the evolutionary relationships) change!

How do we know which tree is the best? Let's look at an analogous example with phenotypic mutations that's easier to follow than genotypic mutations. Consider the two trees below:



tree1

The first tree is a better prediction because it relies on less evolutionary events to happen independent of each other (less "parsimony"). *Think: If you and your roommate are both sitting on the couch eating stawberry popsicles with chia seeds, it's more likely that the popsicles came from the same source than two independent sources.*

Looking at the different trees the algorithms produced, do you have a favorite? (There's no real right answer here... we don't know how these sequences evolved! As long as your can justify your choice, you're good.)

So, what are these models we're switching between? Let's take a look:

12.4.1 Distance Tree Models

Unweighted Pair-Group Method with Arithmetic Averaging (UPGMA)

This model is heavily based upon the assumption that there is an equal rate of evolution (resulting in all branch lengths being the same in the tree). **This is a VERY POOR assumption**, which makes this model rather unreliable.

Neighbor-Joining (NJ)

This model is an agglomerative (bottom-up, or merging) clustering method. We've done agglomerative clustering before *here*, so take a look! Importantly, NJ allows for unequal rates of evolution, which generally improves the output tree.

12.4.2 Distance Calculation Methods

Identity

Distance is equal to the proportion of non matching nucleotides, so lower distance = closer relationship.

Blastn

Matches are worth 5 points, while mismatches are worth -4 points. The formula to calculate the final score is: `1 - (matches*5-mismatches\*4)/(length\*5)`.

Trans

This scoring takes the difference in transitions(purine->purine or pyrimidine->purine) vs transversions(purine->pyrimidine and vice versa) into account. Transversions are less likely to occur, so they are scored -6 compared to the -1 for transitions. Matches are given a score of 6.

12.5 Acknowledgements

Many data and algorithms are adapted from the offical Biopython textbook.Algorithm adapted from Towards Data Science. Diagrams adapted from Khan Academy.

CHAPTER 13

Learning Check 2

13.1 Instructions

13.1.1 Coverage

We assume that you've already covered (and only covered) all of our previous sessions. The techniques for solving the problems below are, with 100% certainty, covered in the previous three lessons.

Note: This does not mean that they are explicitly mentioned in one of our previous lessons. Recall that we emphasize finding specific commands, options, and tools on your own. However, we have covered the core concepts before.

13.1.2 Logistics

The problems below do not specify *how* you should solve them. Instead, we'll present you with tasks to perform and you'll need to determine the output.

Additionally, you'll never need to download any software. Everything you need is on EC2.

13.1.3 Getting Help

In the real world, you're often on your own when you're stuck and need help. Reading through Google searches, documentations, tutorials, and manuals is a real life skill.

However, we don't just want to throw you off the deep end. If you do get stuck on something and you can't figure it out, you can draft an email and send it to Sabeel. Provide a specific description of 1) what you're trying to accomplish 2) what issue you're running into 3) what you've already tried to troubleshoot it. If we find many common issues, we'll send out an FAQ/hints email on Sunday.

13.1.4 Good luck, have fun!

13.2 Problem 1

Find the 100th Fibonacci number (without Googling!).

You may try to write a recursive solution, but it will not run in time. Perhaps this can help?

13.3 Problem 2

In the lab, you're often not going to be given an explanation (such as provided above) of how to perform a task. This challenge will test your ability to use your resources (internet, documentation, etc.) to extend phylogenetic analysis given only input and output.

In: Many times, phylogenetic trees are exported as XML files. XML stands for eXtensible Markup Language, and is used in a wide range of applications that require storage or transport of data in a structured fashion. Provided are two XML files, `single.xml`, containing one phylogenetic tree, and `many.xml`, containing many different phylogenetic trees.

(They'll be downloaded with different names than many or single)many.xmlsingle.xml

Out: When your program is executed, the following should be printed to the terminal for every tree (from both files):

- [Tree Name] or "Unnamed"
- [Tree Description] or "No description"
- [Phylogenetic tree ordered with deepest clades* on top]

*Clade depth is defined by the number of terminal nodes.

13.4 Congratulations! You've completed the introductory portion of the Bioinformatics Crash Course.

CHAPTER 14

About This Course

The Bioinformatics Crash Course is a year-long introductory sequence of lessons intended to introduce new bioinformaticians to practical, lab-applicable skills. For years, it has served as undergraduate and graduate students' first exposure to bioinformatics at UC San Diego.

The course was designed and founded by Sabeel Mansuri and Mark Chernyshev.

It is currently maintained by:

- Ishaan Gupta
- Carleen Li
- Yingxi Lin

It was previously maintained by:

- Mark Chernyshev *(2018-2020)*
- Sabeel Mansuri *(2018-2020)*